ІНФАРМАТЫКА

УДК 004

Э. Н. СЕРЕДИН, Б. А. ЗАЛЕССКИЙ

ФИЛЬТРАЦИЯ И КОРРЕЛЯЦИОННАЯ ОБРАБОТКА ИЗОБРАЖЕНИЙ С ПОМОЩЬЮ ТЕХНОЛОГИИ CUDA

Объединенный институт проблем информатики НАН Беларуси

(Поступила в редакцию 20.02.2015)

Введение. С момента появления компьютеров и до настоящего времени существовало множество задач, которые не могли быть решены с помощью имеющихся вычислительных средств. Приходилось отказываться от реализации многих идей и алгоритмов из-за невозможности проведения экспериментов такими средствами, как персональные компьютеры, специальные процессоры, FPGA (программируемые пользователем вентильные матрицы) и т. д.

Постоянное развитие вычислительной техники, появление все более мощных персональных компьютеров позволило перейти к реализации известных и новых методов. Например, в настоящее время реализованы сложнейшие алгоритмы, строящие и отображающие реалистичные 3D-модели, которые требуют колоссальных объемов вычислений – многих миллиардов операций. Тем не менее время, затрачиваемое для решения некоторых современных задач из области обработки изображений, исчисляется несколькими неделями или даже месяцами, что приводит к необходимости дальнейшего совершенствования аппаратных и программных средств.

Появление новых программируемых видеокарт, доступных для общего пользования, позволяет программам, требующим нескольких часов вычислений на CPU (процессоре), занимать всего несколько минут работы современной GPU (видеокарты). Технологии программирования с использованием видеокарт отличаются от известных подходов и других технологий программирования тем, что видеокарта, в отличие от процессора, обладает сложной параллельной архитектурой, поэтому распараллеливание многих алгоритмов требует больших усилий. Также при выполнении программы на GPU значительную часть времени занимает перемещение данных в памяти, а на вычисления тратится только малая его часть.

В настоящей статье описываются основные концепции технологии CUDA [1], эффективные при компьютерной обработке изображений, и принципы построения программ на основе этой технологии. Приводится сравнительный анализ различных архитектур технологии CUDA с более подробным описанием основных нововведений, таких как динамический параллелизм и возможность нескольким потокам CPU одновременно использовать ядра CUDA одного GPU.

1. Развитие GPU, архитектура и модель программирования CUDA. За последние годы рост тактовой частоты и, следовательно, быстродействие CPU существенно замедлились. Частично проблемы энергопотребления и тепловыделения, ограничивающие быстродействие, удается решить путем уменьшения размеров компонентов микросхем. На сегодняшний день основные производители CPU уже перешли на технологический процесс в 22 нм, но серьезные ограничения, связанные с минимизацией компонентов микросхем, не позволяют и дальше увеличивать производительность только за счет роста частоты, поэтому основной упор делается на увеличении количества параллельно работающих ядер. Так, за последние несколько лет процессоры с двумя и четырьмя ядрами стали доступны широкому кругу потребителей. Даже в мобильных телефонах и планшетных компьютерах уже не редкость наличие двух и более ядер.

Здесь следует упомянуть и суперкомпьютеры. В отличие от многоядерных процессоров, современные суперкомпьютеры построены на многопроцессорной архитектуре, состоящей из многочисленных узлов. Каждый из них представляет собой полноценный многоядерный процессор, связанный с другими для увеличения производительности [2]. Минусом таких систем являются размеры, которые не позволяют использовать их на мобильных объектах и в домашних условиях, а также высокая стоимость, превышающая десятки тысяч долларов.

От упомянутых недостатков избавлены современные графические процессоры (GPU – Graphics Processing Unit), установленные на видеокартах и имеющие сложную параллельную архитектуру. С течением времени они перестали выполнять функции только графических ускорителей и стали мощными программируемыми устройствами, пригодными для решения широкого круга задач. Сегодня GPU – это вычислительное устройство с очень высоким быстродействием, большим объемом собственной памяти и малыми размерами.

Появление программируемых GPU в начале 2000-х гг. позволило решать задачу рендеринга с помощью графических API типа DirectX и OpenGL [3], однако ограничения, накладываемые этими API, стали серьезным препятствием для использования их в неграфических вычислениях. Необходимость изучения специального языка программирования шейдеров, отсутствие средств отладки кода, ограниченность модели программирования препятствовали широкому распространению технологии.

Созданная компанией «Nvidia» технология CUDA (Compute Unified Device Architecture) не использует графических API, но позволяет взаимодействовать с ними. Она является полностью бесплатной и поддерживается всеми видеокартами Nvidia, начиная с GeForce 8-й серии, а также специализированными GPU.

Первая архитектура на базе технологии CUDA под названием **Tesla** была реализована в видеокарте GeForce 8800 GTX. Главное нововведение **Tesla** – это поддержка DirectX 10 и неграфических вычислений при помощи унифицированных шейдеров. Шейдерный конвейер позволяет программе задействовать любое арифметически-логическое устройство на GPU. В архитектуру добавлен набор команд для вычислений общего назначения, разрешен произвольный доступ к ячейкам памяти для чтения и записи, а также к программно-управляемому кэшу.

В архитектуру под названием **Fermi**, впервые появившуюся в видеокарте GeForce 480 GTX, были включены следующие возможности:

– поддержка DirectX 11;

– масштабирование производительности;

– настраиваемый L1 кэш для каждого потокового процессора (отключение L1 кэша эффективно при разреженном доступе к глобальной памяти);

– общий L2 кэш для всех потоковых процессоров;

– атомарные операции [1] в 20 раз быстрее, чем на **Tesla**;

– ECC (коррекция ошибок) [1];

– возможность кэшировать константные значения для блока из глобальной памяти;

– конфликты одновременного доступа к банкам разделяемой памяти возможны на уровне варпа* (в **Tesla** на уровне полуварпа);

– бесконфликтный последовательный доступ к массиву, состоящему из элементов типа double (в отличие от **Tesla);**

– одновременное исполнение нескольких ядер;

– новые инструкции (ballot и др.) [1];

– одновременное копирование CPU → GPU и GPU → CPU.

На смену **Fermi** пришла архитектура под названием **Kepler**, в которой добавлено следующее:

– динамический параллелизм, позволяющий потокам GPU динамически генерировать новые потоки для адаптации к поступающим данным;

– одновременное выполнение до 32 сеток (на **Fermi** – 16);

– выделение для каждого потока отдельной очереди (на **Fermi** все потоки объединяются в одну аппаратную очередь);

* Варп – группа нитей, объединенных на физическом уровне [1].

– параллельность для потоков при отсутствии зависимостей между потоками (на **Fermi** – одновременное выполнение только при переходе от одного потока к другому);

– использование функции **Hyper-Q**, позволяющей нескольким потокам CPU одновременно использовать ядра CUDA одного GPU. Получаемый эффект от использования функции Hyper-Q представлен на рис. 1;

– использование до 255 регистров для каждой нити (в **Tesla** было 128, в **Fermi** – 63), что дает возможность ускорить в несколько раз работу приложений, которым раньше не хватало регистров и поэтому происходили частые сбросы в кэш. Ускорение особенно заметно для приложений с двойной точностью;

– новые высокопроизводительные инструкции (SHFL – обмен данными внутри варпа, ATOM – ускорение расчетов для атомарных операций, SHF – 64-битовый сдвиг, FP32 – деление выполняется быстрее);

– аппаратное ускорение фильтрации 1D-, 2D-, 3D-текстурных данных в 4 раза;

– прямой доступ к текстурному кэшу;

– использование улучшенной реализации ECC.

Графический процессор, построенный на архитектуре **Kepler**, поддерживает: 15 мультипроцессоров, 1,5 Мб L2 кэша, 384-битный канал памяти и PCI Express Gen3, что в итоге обеспечивает производительность более 1 терафлопа в секунду при вычислениях с двойной точностью.

Самой новой на сегодняшний день является архитектура **Maxwell**, к основным нововведениям которой можно отнести следующее:

– реорганизация блоков GPU;

– вычислительная логика располагается в структурах GPC (Graphics Processing Cluster);

– блок GTE (Giga Thread Engine), предназначенный для контроля и одновременного выполнения нескольких ядер, распределения нагрузки между GPC;

– L2 кэш увеличен с 256 до 2048 кб;

– увеличено с 16 до 32 количество блоков на каждый потоковый мультипроцессор;

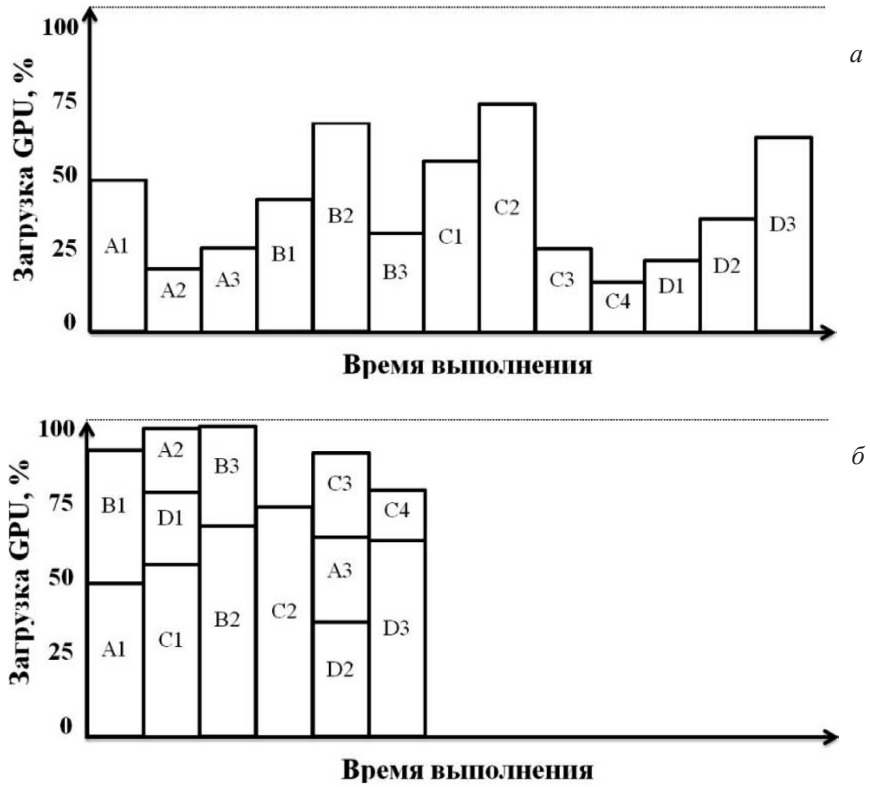

Рис. 1. Загрузка GPU и время работы программы: *а* – без использования Hyper-Q;
б – с использованием Hyper-Q; A1-A3, B1-B3, C1-C4, D1-D3 – запущенные на выполнение потоки разных задач

– уменьшены задержки при выполнении любых арифметических инструкций;

– отдельная специально разделяемая память до 64 кб (в **Fermi** и **Kepler** – распределяется между L1 кэшем и разделяемой памятью);

– общий функционал L1 и текстурного кэшей непосредственно в отдельном блоке;

– новые атомарные операции над 32-битными целыми числами в разделяемой памяти;

– CAS-операции над 32- и 64-битными значениями в разделяемой памяти;

– динамический параллелизм поддерживается во всех видеокартах архитектуры **Maxwell**;

– усовершенствован аппаратный кодек H.264 NVENC (кодирование видео ускорено в 6–8 раз, а декодирование – в 8–10 раз относительно режима работы в реальном времени);

– существенно сниженное энергопотребление за счет внутренней реорганизации потоковых мультипроцессоров.

Развитие аппаратных возможностей видеокарт приведено в табл. 1 на примере первой видеокарты с поддержкой CUDA GeForce 8800 GTX и одной из последних на сегодняшний день GeForce GTX Titan Z.

Таблица 1. **Краткая спецификация видеокарт с поддержкой CUDA**

Видеокарта (архитектура)	GeForce 8800 GTX	GeForce GTX Titan Z
Частота ядра, МГц	575	705 (876)
Частота шейдерного блока, МГц	1350	1750 (7000QDR)
Скорость заполнения текстур, миллиард текселей/с	36,8	338
Количество памяти, Гб	0,768	12
Тип памяти	GDDR3	GDDR5
Интерфейс памяти, бит	384	768 (2 × 384)
Пропускная способность памяти, Гб/с	86,4	672
CUDA Cores, шт.	128	5760
Количество транзисторов, млн шт.	681	14200 (2 × 7100)

К основным преимуществам технологии CUDA можно отнести то, что она изначально была спроектирована для эффективного использования в неграфических вычислениях на GPU. В ней произведена оптимизация обмена данными между CPU и GPU, а также есть доступ к низкоуровневому программированию на PTX-ассемблере.

Технология CUDA реализована как кроссплатформенное программное обеспечение для 32- и 64-битных операционных систем Linux, Mac OS X и Windows.

Для работы с CUDA необходимо наличие графического процессора, поддерживающего архитектуру CUDA, драйвер устройства Nvidia, средства разработки CUDA Development Toolkit и компилятор языка C.

В среду разработки CUDA Development Toolkit входит компилятор nvcc, профилировщик, отладчик GDB для Linux, CUDA Runtime драйвер, руководство по программированию, CUDA Developer SDK, а также несколько стандартных библиотек:

– cuFFT (для вычисления быстрого преобразования Фурье);

– cuBLAS (пакет подпрограмм для линейной алгебры);

– cuSPARS (для работы с разреженными матрицами);

– cuRAND (генератор псевдослучайных чисел);

– NPP (для ускоренной обработки данных) и др.

Для операционных систем Windows компания «Nvidia» выпустила отладчик Parallel Nsight, который может интегрироваться в MS Visual Studio.

В CUDA включены два вида API [4]:

– CUDA Runtime API (высокого уровня);

– CUDA Driver API (низкого уровня).

Одновременное использование обоих CUDA API в одной программе невозможно. При написании программы с использованием Runtime API, все вызовы транслируются в инструкции, обрабатываемые низкоуровневым Driver API. Право выбора вида, используемого API, остается за разработчиком.

Рис. 2. Схема работы программы на CUDA

Программа на CUDA пишется на расширенном языке C (CUDA C), состоящем из традиционного языка C и расширения, включающего в себя спецификаторы функций, спецификаторы переменных, директивы для запуска ядра, встроенные переменные и дополнительные типы данных.

Основная концепция при использовании CUDA состоит в том, что все последовательные вычисления выполняются на CPU, а параллельные – на GPU. При этом код, написанный для GPU, запускается как набор большого числа одновременно работающих нитей (потоков). В отличие от CPU, код для GPU эффективно запускать для тысячи и более нитей, так как на их создание и уничтожение тратится очень малое время.

Алгоритм работы любой программы, написанной на CUDA, можно представить в виде схемы (рис. 2).

На физическом уровне нити разбиваются на группы, называемые варпами (warp). Только нити одного варпа выполняются одновременно, а управление и переключение между варпами осуществляет сам GPU.

Все запущенные нити (threads) объединены в одну сетку (grid), которая состоит из одномерных, двумерных или трехмерных массивов блоков (block). В свою очередь блок состоит из одномерных, двумерных или трехмерных массивов нитей. Каждый блок в сетке имеет свой индекс, аналогично каждая нить – свой индекс внутри блока. Для доступа к конкретной нити используются встроенные переменные. Подобная иерархия позволяет не только удобно работать с многочисленными нитями, но и накладывает ряд ограничений. Например, все нити одного варпа всегда принадлежат одному блоку и могут взаимодействовать между собой только в его пределах.

В программе, написанной для CPU, работа с памятью заключается в выделении и освобождении необходимого количества оперативной памяти, а всю оставшуюся работу на себя берет компилятор. В CUDA же необходимо управлять взаимодействием между оперативной памятью и памятью GPU, а также между различными видами памяти внутри GPU. Физически память видеокарты расположена на потоковых мультипроцессорах и микросхемах. В современных GPU с поддержкой CUDA насчитывается шесть видов памяти [5].

Регистры расположены на потоковых мультипроцессорах. Это самый быстрый вид памяти, доступный на чтение и запись. Хранить в них можно 32-битные целые числа или числа с плавающей точкой. Современные видеокарты располагают 65 536 регистрами на блок, которые распределяются между всеми нитями блока на этапе компиляции. Нити не могут обращаться к регистрам других нитей. Если регистров недостаточно для размещения локальных переменных, то нити начинают использовать локальную память.

Локальная память расположена на микросхемах, не кэшируется и доступна на чтение и запись. Для каждого потокового процессора используется своя локальная память небольшого объема с медленным доступом (порядка 500 тактов).

Разделяемая память расположена на потоковых мультипроцессорах. Доступ к ней такой же быстрый, как и к регистрам. В видеокартах GeForce 6-й серии для каждого блока выделяется

до 48 кб разделяемой памяти, доступной всем нитям своего блока на чтение и запись. От этого количества зависит, сколько блоков может быть одновременно запущено. Разделяемая память используется в виде управляемого кэша первого уровня, помогает снизить задержки при доступе к данным, а также сократить количество обращений к глобальной памяти.

Глобальная память расположена на микросхемах и не кэшируется. В современных видеокартах ее объем достигает нескольких гигабайт. Глобальная память обладает высокой пропускной способностью, но, как и локальная, имеет медленный доступ (порядка 500 тактов). Все нити сетки имеют доступ на чтение и запись к глобальной памяти.

Константная память расположена на микросхемах и доступна только для чтения всеми мультипроцессорами. Кэшируется и имеет объем в 64 кб для видеокарт GeForce 6-й серии. При отсутствии нужных данных в кэше задержка составляет несколько сотен тактов.

Текстурная память расположена на микросхемах и кэшируется. Доступна всем нитям сетки на чтение, а запись в нее может производить только CPU. Выборка данных осуществляется при помощи текстурных блоков, поэтому имеется возможность линейной интерполяции данных без дополнительных затрат. При отсутствии нужных данных в кэше задержка составляет несколько сотен тактов.

Хотя глобальная, локальная, текстурная и константная память являются памятью видеокарты, их отличия проявляются в виде различных алгоритмов кэширования и моделях доступа. Например, процессор может работать только с глобальной, константной и текстурной памятью. Понимание особенностей использования каждой из этих видов памяти, а также их взаимодействия между собой играет важную роль в достижении наилучших результатов с точки зрения быстродействия.

2. Оценка эффективности технологии CUDA в задачах обработки изображений. Технология программирования CUDA широко используется для решения задач в различных областях, например в компьютерном зрении и обработке изображений, биоинформатике и биологических науках, экономике, вычислительной аэро- и гидродинамике, анализе данных, физике, медицине и т. д. [6–11].

Применение CUDA позволяет ускорить вычисления от нескольких раз до нескольких порядков по сравнению с процессорной реализацией функции, алгоритма или метода [12]. В качестве примера оценим прирост быстродействия, получаемый с помощью CUDA, при решении некоторых задач в области обработки изображений.

2.1. Линейные оконные фильтры. Линейные оконные фильтры задаются функцией – ядром фильтра, которое отлично от нуля только в некоторой окрестности для каждой точки, а за пределами этой окрестности равно нулю. Далее фильтрация производится при помощи дискретной свертки и результатом является новое изображение. Обработку пикселов, которые находятся в окрестностях краев изображения, можно производить несколькими способами, в зависимости от требуемых условий (например, определить значения пикселов за границами изображения при помощи экстраполяции или зеркального продолжения изображения и т. д.).

Универсальным линейным оконным фильтром является фильтр с произвольно-задаваемыми значениями его ядра. При программной реализации универсального линейного оконного фильтра на CUDA было определено, что лучшую производительность дает следующий алгоритм.

Шаг 1. Создаются три линейных массива в памяти видеокарты:

cudaSource – данные изображения;

cudaMask – данные применяемого универсального линейного фильтра;

cudaResult – результирующее изображение.

Шаг 2. Копируются в cudaSource и cudaMask соответствующие данные из массивов в оперативной памяти.

Шаг 3. Запускается ядро программы на CUDA со следующими параметрами:

```
N = 16;
grid(widthSource / N + 1, heightSource / N + 1)
threads(N, N)
Kernel<<<grid, threads>>> (…)
```

где N – целое значение, которое выбрано на основе проведенных тестов и рекомендаций других разработчиков;

widthSource – ширина изображения;
heightSource – высота изображения;
grid – конфигурация блоков на сетке;
threads – конфигурация нитей в блоке.

Шаг 4. На видеокарте задается 2D-индексация нитей и блоков, для которой вычисления в каждом пикселе выполняются отдельной нитью. Использование индексации, где каждая нить выполняет вычисления в нескольких точках, дает схожие результаты производительности.

Шаг 5. На видеокарте производятся вычисления для каждого пиксела в параллельном режиме. Результирующие данные записываются в cudaResult.

Шаг 6. Копируются данные из cudaResult в массив, находящийся в оперативной памяти.

Сравнительные результаты быстродействия приведены в табл. 2.

Таблица 2. **Время расчета линейного оконного фильтра для различных CPU и GPU, мс**

Устройство	Разрешение изображения и размеры окна фильтра, пикселы					
	720 × 480		1920 × 1080		5000 × 5000	
	3 × 3	5 × 5	3 × 3	5 × 5	3 × 3	5 × 5
CPU Core2Duo E6550 2,33 ГГц (2)*	12,5	15,5	31,2	68,8	389,0	828,2
CPU Core2Quad Q6600 2,66 ГГц (4)*	6,3	12,4	35,8	65,6	254,1	522,6
CPU i7-4770K 3,50 ГГц (8)*	1,5	3,1	7,7	15,6	84,5	164,2
GPU GTX 550Ti	3,8	4,0	15,4	17,3	171,4	187,8
GPU GTX 650Ti	3,4	3,8	11,3	13,8	118,0	149,0
GPU GTX 750Ti	3,0	3,3	12,6	15,9	138,5	158,2

* В скобках указано количество ядер CPU, задействованных при вычислениях.

2.2. Нелинейные оконные фильтры (на примере медианы). В нелинейных оконных фильтрах вокруг каждого пиксела строится окно заранее заданных размеров. Рассчитывается результирующее значение для каждого пиксела, в зависимости от применяемого фильтра. Одним из самых известных и используемых нелинейных фильтров является медиана. При медианной фильтрации элементы окна каждого пиксела упорядочиваются по возрастанию или убыванию. Результирующим значением является центральный элемент полученной упорядоченной последовательности. Обработку пикселов, которые находятся в окрестностях краев изображения, можно производить так же, как описано ранее для линейных фильтров. При программной реализации медианного фильтра на CUDA было определено, что лучшую производительность дает следующий алгоритм.

Шаг 1. Создаются два линейных массива в памяти видеокарты:
cudaSource – данные изображения;
cudaResult – результирующее изображение.

Шаг 2. Копируются в cudaSource соответствующие данные из массива в оперативной памяти.

Шаг 3. Запускается ядро программы на CUDA со следующими параметрами:
N = 16;
grid(widthSource / N + 1, heightSource / N + 1)
threads(N, N)
Kernel<<<grid, threads,sizeSharedData>>> (…)
где N – целое значение, которое выбрано на основе проведенных тестов и рекомендаций других разработчиков;
widthSource – ширина изображения;
heightSource – высота изображения;
grid – конфигурация блоков на сетке;
threads – конфигурация нитей в блоке;
sizeSharedData – размер дополнительно выделяемой разделяемой памяти для каждого блока.

Шаг 4. На видеокарте задается 2D-индексация нитей и блоков, для которой вычисления в каждом пикселе выполняются отдельной нитью. Дополнительно выделенная разделяемая память

доступна для всех запущенных нитей внутри блока, поэтому произведена еще одна индексация для того, чтобы каждая нить использовала только свою часть этой разделяемой памяти.

Шаг 5. На видеокарте производятся вычисления для каждого пиксела в параллельном режиме. Результирующие данные записываются в cudaResult.

Шаг 6. Копируются данные из cudaResult в массив, находящийся в оперативной памяти.

Сравнительные результаты быстродействия приведены в табл. 3.

Таблица 3. **Время расчета нелинейного оконного фильтра (медиана) для различных CPU и GPU, мс**

Устройство	Разрешение изображения и размеры окна фильтра, пикселы					
	720 × 480		1920 × 1080		5000 × 5000	
	3 × 3	5 × 5	3 × 3	5 × 5	3 × 3	5 × 5
CPU Core2Duo E6550 2,33 ГГц (2)*	36,1	181,4	179,7	920,3	2235,9	10821,9
CPU Core2Quad Q6600 2,66 ГГц (4)*	15,6	103,1	106,2	516,4	1262,0	6350,9
CPU i7-4770K 3,50 ГГц (8)*	12,5	45,2	39,2	192,3	414,2	2292,2
GPU GTX 550Ti	3,5	14,3	16,1	70,1	178,7	833,7
GPU GTX 650Ti	3,0	8,9	12,7	45,5	141,1	467,2
GPU GTX 750Ti	2,4	6,4	10,9	31,7	124,5	343,0

* В скобках указано количество ядер CPU, задействованных при вычислениях.

2.3. Корреляция. Один из самых надежных методов поиска заданного объекта (эталон) на изображении (шаблон) основан на сравнении оконных корреляций Пирсона [13] между эталоном и всеми соответствующими областями шаблона. Суть метода заключается в следующем: окно, размер которого соответствует эталону, передвигается пиксел за пикселом по шаблону. Для каждого положения окна вычисляется коэффициент корреляции (рис. 3). Позиция, для которой коэффициент корреляции достигает своего наивысшего значения, берется в качестве координат лучшего соответствия.

Известно, что вычисление корреляции требует больших вычислительных затрат. Для уменьшения времени расчета корреляции на CPU зачастую используют быстрое преобразование Фурье, а также другие методы [13]. Еще одним способом является уменьшение количества точек, в которых вычисляется корреляция, за счет применения дополнительных алгоритмов [14]. Все это позволяет значительно уменьшить время вычислений, но не дает возможности добиться нужного быстродействия. Например, при решении задачи сопровождения объекта, наблюдаемого видеокамерой, в режиме реального времени на обработку одного кадра отводится 25–35 мс. Ни одна из известных реализаций корреляционных алгоритмов не позволяет решить задачу за указанное время. Если же для решения задачи требуется дополнительно учитывать повороты и масштабирование изображения, то от использования корреляции и вовсе приходится отказываться.

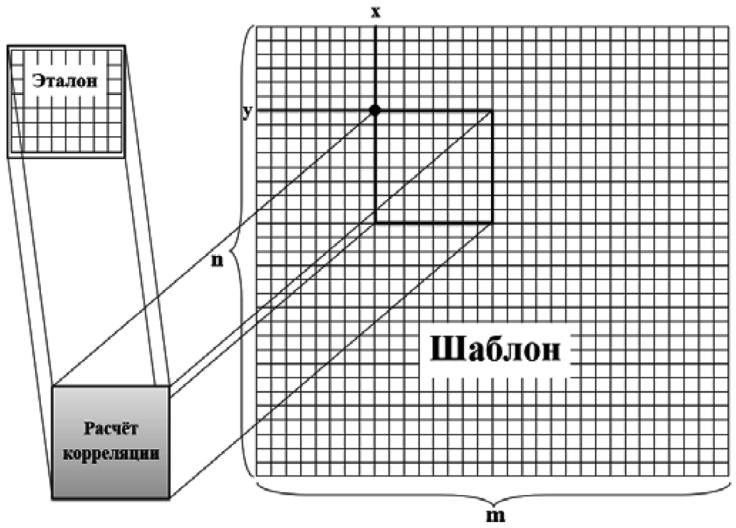

Рис. 3. Пример расчета корреляции Пирсона в точке (*x, y*)

Ситуация изменилась с появлением видеокарт от «Nvidia» с поддержкой архитектур **Kepler** и **Maxwell** (видеокарты GeForce 6-й серии и более новые). Использование технологии CUDA позволяет произвести вычисление корреляции Пирсона во всех точках шаблона без каких-либо дополнительных действий с изображением.

При программной реализации корреляции Пирсона на CUDA было определено, что лучшую производительность дает следующий алгоритм.

Шаг 1. Формируются текстуры в памяти видеокарты: одна для шаблона (cudaSource) и одна для эталона (cudaTemplate).

Шаг 2. Создаются линейные массивы в памяти видеокарты cudaResult для значений корреляции Пирсона и cudaResultIndex для значений угла поворота эталона.

Шаг 3. Копируются данные шаблона в cudaSource, а втекстуру cudaTemplate помещаются значения всех вариантов поворота эталона (например, как на рис. 4).

Шаг 4. Запускается ядро программы на CUDA со следующими параметрами:

N = 16;

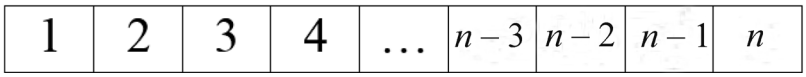

Рис. 4. Пример формирования текстуры cudaTemplate для всех вариантов поворота эталона, где *n* – количество поворотов эталона

grid(widthSource / N + 1, heightSource / N + 1, countAngles)

threads(N, N)

Kernel<<<grid, threads>>> (…)

где N – целое значение, которое выбрано на основе проведенных тестов и рекомендаций других разработчиков;

widthSource – ширина изображения;

heightSource – высота изображения;

countAngles – количество поворотов эталона;

grid – конфигурация блоков на сетке;

threads – конфигурация нитей в блоке.

Шаг 5. На видеокарте задается 3D-индексация блоков и 2D-индексация нитей, для которой вычисления в каждом пикселе всех вариантов поворота эталона выполняются отдельной нитью.

Шаг 6. На видеокарте производятся вычисления корреляции Пирсона для каждого пиксела всех вариантов поворота эталона в параллельном режиме. В cudaResult записываются максимальные значения корреляции для каждого пиксела, а в cudaResultIndex – значения угла поворота эталона с максимальной корреляцией.

Шаг 7. Копируются данные из cudaResult и cudaResultIndex в массивы, находящиеся в оперативной памяти.

Сравнительные результаты быстродействия приведены в табл. 4, 5.

Таблица 4. **Время расчета корреляции Пирсона для различных CPU и GPU для 60 поворотов эталона, с**

Устройство	Разрешение изображения и размеры окна фильтра, пикселы					
	720 × 480		1920 × 1080		5000 × 5000	
	8 × 8	32 × 32	8 × 8	32 × 32	8 × 8	32 × 32
CPU Core2Duo E6550 2,33 ГГц (2)*	18,8	240,9	94,0	127,4	1147,3	15933,8
CPU Core2Quad Q6600 2,66 ГГц (4)*	9,0	117,0	45,6	617,0	555,1	7689,9
CPU i7-4770K 3,50 ГГц (8)*	1,1	11,6	5,6	63,0	68,9	788,9
GPU GTX 550Ti	0,11	2,4	0,55	7,8	6,6	98,1
GPU GTX 650Ti	0,07	0,85	0,35	4,5	4,3	56,1
GPU GTX 750Ti	0,08	1,0	0,37	5,3	4,5	66,5

* В скобках указано количество ядер CPU, задействованных при вычислениях.

Таблица 5. **Время расчета корреляции Пирсона для различных CPU и GPU для 360 поворотов эталона, с**

Устройство	Разрешение изображения и размеры окна фильтра, пикселы					
	720 × 480		1920 × 1080		5000 × 5000	
	8 × 8	32 × 32	8 × 8	32 × 32	8 × 8	32 × 32
CPU Core2Duo E6550 2,33 ГГц (2)*	111,6	1444,0	565,8	7668,1	6887,2	≈87186
CPU Core2Quad Q6600 2,66 ГГц (4)*	54,3	702,3	274,4	3706,9	3333,6	≈42407
CPU i7-4770K 3,50 ГГц (8)*	6,7	70,3	33,8	378,1	418,1	≈4352
GPU GTX 550Ti	0,64	9,9	3,2	47,1	39,0	589,0
GPU GTX 650Ti	0,42	5,1	3,0	26,9	25,4	336,2
GPU GTX 750Ti	0,45	7,0	2,2	31,9	26,4	398,6

* В скобках указано количество ядер CPU, задействованных при вычислениях.

Для сравнения быстродействия разработанных алгоритмов были выбраны несколько стандартных разрешений. При написании программного обеспечения и для всех тестов использовались: операционная система Windows 8.1 64bit, CUDA версии 6.5, драйвер видеокарты 347.25, MS Visual Studio 2013, язык программирования C++ и тип данных float (с плавающей запятой одинарной точности). При расчетах на видеокарте не применялись какие-либо дополнительные приемы оптимизации (использование паттернов доступа к разделяемой памяти, эффективных алгоритмов редукции и т. д.), а для CPU расчет производился методом прохода по всем точкам с распараллеливанием на максимальное количество поддерживаемых ядер каждым из процессоров. Время, затраченное на написание программного обеспечения, примерно одинаковое.

Из табл. 2–5 видно, что использование бюджетных видеокарт уменьшает время вычислений на несколько порядков. При этом, в отличие от реализации алгоритмов на процессоре, где загрузка достигает 100 %, при вычислениях на видеокарте ресурсы процессора остаются свободными и могут использоваться для решения других задач. Можно также отметить, что в некоторых случаях видеокарта с более старой архитектурой CUDA показывает лучшую производительность, что связано с аппаратными возможностями видеокарт и степенью оптимизации алгоритма под используемую архитектуру CUDA. При относительно небольшом количестве вычислений процессор i7 с работающими 8 ядрами не сильно уступает бюджетным видеокартам и иногда даже обгоняет их. Однако непосредственно сами вычисления на видеокарте занимают всего 33–50 % от общего времени работы, остальное уходит на копирование данных между памятью CPU и GPU и наоборот. Поэтому использование более производительных видеокарт может ускорить вычисления еще до 2–3 раз только за счет уменьшения времени копирования данных.

Заключение. Представлен краткий обзор появившихся в последнее время архитектур видеокарт от «Nvidia», таких как **Tesla, Fermi, Kepler, Maxwell**, новых возможностей и особенностей технологии программирования CUDA, ставших доступными в этих архитектурах. Акцент сделан на использование CUDA для обработки изображений. На примерах продемонстрировано, что использование GPU при решении актуальных задач обработки изображений позволяет уменьшить на несколько порядков время выполнения программ по сравнению с их параллельными версиями, реализованными на CPU.

В дальнейшем планируется разработать быстрые реализации алгоритмов обработки и распознавания изображений, требующие выполнения в режиме реального времени, с использованием новых возможностей технологии CUDA.

Литература

1. https://developer.nvidia.com/category/zone/cuda-zone [Электронный ресурс].
2. *Таненбаум Э.* Архитектура компьютера. СПб., 2007.
3. *Евченко А. И.* OpenGL и DirectX: программирование графики. СПб., 2006.
4. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html [Электронный ресурс].
5. *Боресков А. В*. Основы работы с технологией CUDA. М., 2010.
6. http://www.nvidia.ru/object/gpu-computing-applications-ru.html [Электронный ресурс].
7. *Залесский Б. А., Середин Э. Н.* // Информатика. 2014. № 41.
8. *Josth R.* // J. of Real-Time Image Processing. 2011. Vol. 7.

9. *Joaquı´n F*. // J. of Real-Time Image Processing. 2011. Vol. 7.
10. *Yoon-Seok Choi*. // J. of Real-Time Image Processing. 2014. Vol. 9.
11. *Gembris D*. // J. of Real-Time Image Processing. 2010.
12. *Сандерс Дж.* Технология CUDA в примерах: введение в программирование графических процессоров. М., 2011.
13. *Гонсалес Р*. Цифровая обработка изображений. М., 2005.
14. *Сойфер В. А*. Методы компьютерной обработки изображений. М., 2003.

E. N. SEREDIN, B. A. ZALESKY

FUILTRATION AND CORRELATION PROCESSING OF IMAGES BY THE CUDA TECHNOLOGY

Summary

The basic concepts and specificity of the programming technology of CUDA video cards are presented. The efficiency of the technology is demonstrated on image processing tasks. Results of a comparative performance analysis of program implementations on the GPU and CPU are adduced for urgent tasks of image processing. It is shown that CUDA allows accelerating computations of image processing tasks by several orders of magnitude. In particular, the use of the CUDA technology has made possible to implement correlation algorithms for tracking objects on video sequences in real time.